Evaluation of the runtime performance of control flow structures for dynamic dispatch in Java

Olivier Zendra, Karel Driesen,
INRIA / McGill University
School of Computer Science - ACL Group
Montreal, Quebec, Canada H3A 2A7
{ozendra,karel}@cs.mcgill.ca

Feng Qian and Laurie Hendren
McGill University
School of Computer Science - Sable Group
Montreal, Quebec, Canada H3A 2A7
{fqian,hendren}@cs.mcgill.ca

ABSTRACT

We present an ongoing study of control flow structures in Java. We have designed a collection of benchmarks for various JVM and hardware platforms, to characterize the performance of these structures when used to simulate dynamic dispatch implementations.

Keywords

Java, dynamic dispatch, control structure, optimization, JVM

1. INTRODUCTION

Dynamic dispatch (or virtual method invocation) in object-oriented languages, especially Java, are frequent and thus have to be cheap, to avoid incurring a performance penalty at runtime.

We study the performance of various implementations of dynamic dispatch in Java by using several control flow structures at the Java source-code and byte-code level, in order to evaluate them across a wide range of JVMs. We propose and demonstrate a methodology which relies on micro-kernel benchmarking to determine the relative performance of different control flow instructions on several JVMs and hardware platforms.

This paper is organized as follows. Section 2 reviews dynamic dispatch, the issues at stake and current common solutions. Section 3 briefly presents the experimental setup we designed to produce the results provided in the poster. Section 4 concludes and indicates future directions for our ongoing work.

2. MOTIVATION AND BACKGROUND

Virtual method invocations in Java are frequent (every 12 to 40 byte code executions in SPECJVM98 [4]). They are also expensive because the target method depends on the run-time type of the receiver, which generally can't be determined until actual execution.

Hardware Solutions. In native code, virtual method invocations translate to two dependent loads followed by an indirect branch. The latter is responsible for most of the call overhead.

Branches are expensive on modern, deeply pipelined processors because the next instruction cannot be fetched with certainty until the branch is resolved, typically at a late stage in the pipeline (e.g. after 10 cycles on a Pentium III, 20 on a Pentium 4). Most processors try to avoid these "pipeline bubbles" by *speculatively executing* instructions of the most likely execution path, as predicted by separate branch prediction micro-architecture. Sophisticated predictors [2, 3] can reach high prediction rates, but generally require large on-chip structures.

Indirect branches are more difficult to predict than conditional ones. A conditional branch has only one target, encoded in the instruction itself as an offset, so a processor only needs to predict whether the conditional branch is taken or not (one bit). Indirect branches can have many different targets and therefore require prediction of the complete target address (32 or 64 bits). Indirect branch predictors (Branch Target Buffers — BTB) thus tend to be smaller than conditional branch predictors (Branch History Buffers — BHT). Unfortunately, even a hypothetical unlimited BTB mispredicts 25% of all indirect branches on a suite of large object-oriented programs [2].

Software Solutions. Most JVMs include some way to devirtualize method invocations which are actually monomorphic, by replacing the costly polymorphic call sequence by a direct jump. Various forms of whole program analysis (e.g. [1, 5]) show that most invocations in Java are monomorphic.

Some JVMs use a dynamic approach. HotSpot™ relies on a form of inline caching. The first time a virtual method invocation is executed, it is replaced by a direct call preceded by a type check. Subsequent executions with the same target are thus direct, whereas executions with a different target fall back to a standard virtual function call.

Actual run-time polymorphism can also be optimized in software, for example by using Binary Tree Dispatch (BTD), as implemented in SmallEiffel [6]. BTD replaces a sequence of powerful dispatch instructions using an indirect branch by a sequence of simpler instructions (conditional branches and direct calls). When the sequence of simple instructions remains small, it can be more efficient than a call through a virtual function table, and should perform particularly well on processors with accurate conditional branch prediction and large BHT.

Motivation. Given the large variety of JVM implementations and hardware platforms them on, we expect that the performance of method invocations differs substantially between execution environments. In addition, the performance of truly polymorphic invocations is also determined by the

order in which different targets occur at run time. A particular call site may indeed switch often between a limited number of targets and thus be less efficient than a call site which has many targets but does not switch often.

We aim to measure the performance of various control structures in Java, under a variety of different degrees of run-time polymorphism, in order to determine how call sites can be optimized. For example, if a particular call site alternates between two targets, an indirect branch call sequence may show particularly low performance, because the branch target buffer stores the last target and therefore misses every time. Such a call site is better implemented by a sequence of two conditional branches, since both targets will then be stored in the BTB. History pattern-based conditional branch prediction, such as in the Pentium III, is then able to capture the regularity of the conditional branches, resulting in 100% accurate prediction.

3. EXPERIMENTAL SETUP

We try to answer three questions. First, how is the relative performance of different control structures influenced by hardware? Second, how is it affected by the JVM? Third, what's the influence of the degree and the pattern of polymorphism?

We aim to find whether some specific control structures can be used to optimize — under specific execution conditions but across all VMs and platforms — virtual method invocation. Also, we want to find whether some reasonable assumptions (e.g direct static calls being faster than indirect ones) actually hold in practice.

We therefore designed a number of micro benchmarks which simulate various dynamic dispatch implementations (switch statements, sequences of ifs, and corresponding virtual method invocations). All benchmarks use the same superstructure: a long-running iterating multiple times over a large array. The latter is initialized with a variety of object types, representing different patterns and degrees of polymorphism. This combination of structures and patterns provides us with about 5000 different measurements in the testing suite. We then run this suite across several different JVM/platform combinations. In summary, we measure the performance of control structures within three dimensions: hardware, virtual machine, and dynamic program behavior.

The poster shows the essence of this large amount of data. Preliminary results indicate that execution patterns can drastically influence the cost of control structures whose cost was expected to be identical. For example, on at least one JVM, table switches are not a good alternative for virtual method calls, since the execution cost goes up linearly with the rank of the executed case. On the other hand, short sequences of if statements are more efficient than virtual method calls on the same JVM, indicating that byte-code level portable optimizations are possible for highly polymorphic virtual method calls with a limited number of targets.

4. CONCLUSIONS AND FUTURE WORK

The implementation of dynamic dispatch is very important for object-oriented program performance. A number of optimization techniques exist, aimed at de-virtualizing polymorphic calls which can be determined, either at compile-time or runtime, to be actually monomorphic. Complementary techniques, either software- or hardware-based, seek to optimize actual run-time polymorphism as well.

We present an ongoing study of various control flow structures for dispatch in Java, varying hardware, virtual machine and execution patterns.

The results are important to Java compiler and Java VM implementers, especially when implementing multiple-target control structures such as dispatch. These results are also useful to Java developers, since they stress some differences between the various JVMs, highlighting strengths to take advantage of and weaknesses to avoid.

We plan to continue our work by implementing platform-independent control structure optimizations. We also intend to continue to expand our benchmark suite in four dimensions, by including more control structures, different hardware, several virtual machines, and various real-life execution traces.

Additional information may be found at the Adaptive Computation Laboratory: `http://www.cs.mcgill.ca/acl`.

Acknowledgements

We thank Matt Holly who reviewed early versions of this abstract. We would also like to acknowledge support from NSERC and FCAR (Canada), and INRIA (France).

5. REFERENCES

[1] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 324–341. ACM Press, 1996.

[2] K. Driesen and U. Hölzle. Accurate Indirect Branch Prediction. In *1998 International Symposium on Computer Architecture (ISCA'98)*, July 1998.

[3] K. Driesen and U. Hölzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Micro'98 Conference*, pages 249–258, Dec. 1998.

[4] K. Driesen, P. Lam, J. Miecznikowski, F. Qian, and D. Rayside. On the Predictability of Invoke Targets in Java Byte Code. In *2nd Annual Workshop on Hardware Support for Objects and Microarchitectures for Java*, pages 6–10, Sept. 2000.

[5] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical Virtual Method Call Resolution for Java. In *15th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, volume 35, pages 264–280. ACM Press, Oct. 2000.

[6] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, pages 125–141. ACM Press, Oct. 1997.